Normally, there are no other ways to modify the data of an existing smart contract. If the code of the smart contract does not provide any ways to modify the persistent data (e.g., if it is a simple wallet smart contract as described in **1.7.6**, which initializes the persistent data with the user's public key and does not intend to ever change it), then it will be effectively immutable—unless the code of the smart contract is modified first.

1.8.2. Modification of the code of a smart contract. Similarly, the code of an existing smart contract may be modified only if some provisions for such an upgrade are present in the current code. The code is modified by invoking TVM primitive `SETCODE`, which sets the root of the code for the current smart contract from the top value in the TVM stack. The modification is applied only after the normal termination of the current transaction.

Typically, if the developer of a smart contract wants to be able to upgrade its code in the future, she provides a special "code upgrade method" in the original code of the smart contract, which invokes `SETCODE` in response to certain inbound "code upgrade" messages, using the new code sent in the message itself as an argument to `SETCODE`. Some provisions must be made to protect the smart contract from unauthorized replacement of the code; otherwise, control of the smart contract and the funds on its balance could be lost. For example, code upgrade messages might be accepted only from a trusted source address, or they might be protected by requiring a valid cryptographic signature and a correct sequence number.

1.8.3. Keeping the code or data of the smart contract outside the blockchain. The code or data of the smart contract may be kept outside the blockchain and be represented only by their hashes. In such cases, only empty inbound messages may be processed, as well as messages carrying a correct copy of the smart-contract code (or its portion relevant for processing the specific message) and its data inside special fields. An example of such a situation is given by the uninitialized smart contracts and constructor messages described in **1.7**.

1.8.4. Using code libraries. Some smart contracts may share the same code, but use different data. One example of this is wallet smart contracts (cf. **1.7.6**), which are likely to use the same code (throughout all wallets created by the same software), but with different data (because each wallet must use its own pair of cryptographic keys). In this case, the code for all the wallet smart contracts is best committed by the developer into a shared

library; this library would reside in the masterchain, and be referred to by its hash using a special "external library cell reference" as the root of the code of each wallet smart contract (or as a subtree inside that code).

Notice that even if the library code becomes unavalable—for example, because its developer stops paying for its storage in the masterchain—it is still possible to use the smart contracts referring to this library, either by committing the library again into the masterchain, or by including its relevant parts inside a message sent to the smart contract. This external cell reference resolution mechanism is discussed in more detail later in **4.4.3**.

1.8.5. Destroying smart contracts. Notice that a smart contract cannot really be destroyed until its balance becomes zero or negative. It may become negative as a result of collecting persistent storage payments, or after sending a value-bearing outbound message transferring almost all of its previous balance.

For example, a user may decide to transfer all remaining funds from her wallet to another wallet or smart contract. This may be useful, for instance, if one wants to upgrade the wallet, but the wallet smart contract does not have any provisions for future upgrades; then one can simply create a new wallet and transfer all funds to it.

1.8.6. Frozen accounts. When the balance of an account becomes non-positive after a transaction, or smaller than a certain workchain-dependent minimum, the account is *frozen* by replacing all its code and data by a single 32-byte hash. This hash is kept afterwards for some time (e.g., a couple of months) to prevent recreation of the smart contract by its original creating transaction (which still has the correct hash, equal to the account address), and to allow its owner to recreate the account by transferring some funds and sending a message containing the account's code and data, to be reinstated in the blockchain. In this respect, frozen accounts are similar to uninitialized accounts; however, the hash of the correct code and data for a frozen account is not necessarily equal to the account address, but is kept separately.

Notice that frozen accounts may have a negative balance, indicating that persistent storage payments are due. An account cannot be unfrozen until its balance becomes positive and larger than a prescribed minimum value.

2 Message forwarding and delivery guarantees

This chapter discusses the forwarding of messages inside the TON Blockchain, including the Hypercube Routing (HR) and Instant Hypercube Routing (IHR) protocols. It also describes the provisions required to implement the message delivery guarantees and the FIFO ordering guarantee.

2.1 Message addresses and next-hop computation

This section explains the computation of transit and next-hop addresses by the variant of the hypercube routing algorithm employed in TON Blockchain. The hypercube routing protocol itself, which uses the concepts and next-hop address computation algorithm introduced in this section, is presented in the next section.

2.1.1. Account addresses. The *source address* and *destination address* are always present in any message. Normally, they are *(full) account addresses*. A full account address consists of a *workchain_id* (a signed 32-bit big-endian integer defining a workchain), followed by a (usually) 256-bit *internal address* or *account identifier account_id* (which may also be interpreted as an unsigned big-endian integer) defining the account within the chosen workchain.

Different workchains may use account identifiers that are shorter or longer than the "standard" 256 bits used in the masterchain ($workchain\_id = -1$) and in the basic workchain ($workchain\_id = 0$). To this end, the masterchain state contains a list of all workchains defined so far, along with their account identifier lengths. An important restriction is that the *account_id* for any workchain must be at least 64 bits long.

In what follows, we often consider only the case of 256-bit account addresses for simplicity. Only the first 64 bits of the *account_id* are relevant for the purposes of message routing and shardchain splitting.

2.1.2. Source and destination addresses of a message. Any message has both a *source address* and a *destination address*. Its source address is the address of the account (smart contract) that has created the message while processing some transaction; the source address cannot be changed or set arbitrarily, and smart contracts heavily rely on this property. By contrast, when a message is created, any well-formed destination address may be chosen; after that, the destination address cannot be changed.

2.1.3. External messages with no source or destination address. Some messages can have no source or no destination address (though at least one of them must be present), as indicated by special flags in the message header. Such messages are the *external messages* intended for the interaction of the TON Blockchain with the outside world—human users and their cryptowallet applications, off-chain and mixed applications and services, other blockchains, and so on.

External messages are never routed inside the TON Blockchain. Instead, "messages from nowhere" (i.e., with no source address) are directly included into the *InMsgDescr* of a destination shardchain block (provided some conditions are met) and processed by a transaction in that very block. Similarly, "messages to nowhere" (i.e., with no TON Blockchain destination address), also known as *log messages*, are also present only in the block containing the transaction that generated such a message.[12]

Therefore, external messages are almost irrelevant for the discussion of message routing and message delivery guarantees. In fact, the message delivery guarantees for outbound external messages are trivial (at most, the message must be included into the *LogMsg* part of the block), and for inbound external messages there are none, since the validators of a shardchain block are free to include or ignore suggested inbound external messages at their discretion (e.g., according to the processing fee offered by the message).[13]

In what follows, we focus on "usual" or "internal" messages, which have both a source and a destination address.

2.1.4. Transit and next-hop addresses. When a message needs to be routed through intermediate shardchains before reaching its intended destination, it is assigned a *transit address* and a *next-hop address* in addition to the (immutable) source and destination addresses. When a copy of the

[12]"Messages to nowhere" may have some special fields in their body indicating their destination outside the TON Blockchain—for instance, an account in some other blockchain, or an IP address and port—which may be interpreted by the third-party software appropriately. Such fields are ignored by the TON Blockchain.

[13]The problem of bypassing possible validator censorship—which could happen, for instance, if all validators conspire not to include external messages sent to accounts belonging to some set of blacklisted accounts—is dealt with separately elsewhere. The main idea is that the validators may be forced to promise to include a message with a known hash in a future block, without knowing anything about the identity of the sender or the receiver; they will have to keep this promise afterwards when the message itself with pre-agreed hash is presented.

message resides inside a transit shardchain awaiting its relay to its next hop, the *transit address* is its intermediate address lying in the transit shardchain, as if belonging to a special message-relay smart contract whose only job is to relay the unchanged message to the next shardchain on the route. The *next-hop address* is the address in a neighboring shardchain (or, on some rare occasions, in the same shardchain) to which the message needs to be relayed. After the message is relayed, the next-hop address usually becomes the transit address of the copy of the message included in the next shardchain.

Immediately after an outbound message is created in a shardchain (or in the masterchain), its transit address is set to its source address.[14]

2.1.5. Computation of the next-hop address for hypercube routing. The TON Blockchain employs a variant of hypercube routing. This means that the next-hop address is computed from the transit address (originally equal to the source address) as follows:

1. The (big-endian signed) 32-bit *workchain_id* components of both the transit address and destination address are split into groups of n_1 bits (currently, $n_1 = 32$), and they are scanned from the left (i.e., the most significant bits) to the right. If one of the groups in the transit address differs from the corresponding group in the destination address, then the value of this group in the transit address is replaced by its value in the destination address to compute the next-hop address.

2. If the *workchain_id* parts of the transit and destination addresses match, then a similar process is applied to the *account_id* parts of the addresses: The *account_id* parts, or rather their first (most significant) 64 bits, are split into groups of n_2 bits (currently, $n_2 = 4$ bit groups are used, corresponding to the hexadecimal digits of the address) starting from the most significant bit, and are compared starting from the left. The first group that differs is replaced in the transit address with its value in the destination address to compute the next-hop address.

3. If the first 64 bits of the *account_id* parts of the transit and destination addresses match as well, then the destination account belongs to the current shardchain, and the message should not be forwarded

[14]However, the internal routing process described in **2.1.11** is applied immediately after that, which may further modify the transit address.

outside the current shardchain at all. Instead, it must be processed by a transaction inside it.

2.1.6. Notation for the next-hop address. We denote by

$$\text{NextHop}(\xi, \eta) \tag{13}$$

the next-hop address computed for current (source or transit) address ξ and destination address η.

2.1.7. Support for anycast addresses. "Large" smart contracts, which can have separate instances in different shardchains, may be reached using *anycast destination addresses*. These addresses are supported as follows.

An anycast address (η, d) consists of a usual address η along with its "splitting depth" $d \leq 31$. The idea is that the message may be delivered to any address differing from η only in the first d bits of the internal address part (i.e., not including the workchain identifier, which must match exactly). This is achieved as follows:

- The effective destination address $\tilde{\eta}$ is computed from (η, d) by replacing the first d bits of the internal address part of η with the corresponding bits taken from the source address ξ.
- All computations of $\text{NextHop}(\nu, \eta)$ are replaced by $\text{NextHop}(\nu, \tilde{\eta})$, for $\nu = \xi$ as well as for all other intermediate addresses ν. In this way, Hypercube Routing or Instant Hypercube Routing will ultimately deliver the message to the shardchain containing $\tilde{\eta}$.
- When the message is processed in its destination shardchain (the one containing address $\tilde{\eta}$), it may be processed by an account η' of the same shardchain differing from η and $\tilde{\eta}$ only in the first d bits of the internal address part. More precisely, if the common shard address prefix is s, so that only internal addresses starting with binary string s belong to the destination shard, then η' is computed from η by replacing the first $\min(d, |s|)$ bits of the internal address part of η with the corresponding bits of s.

That said, we tacitly ignore the existence of anycast addresses and the additional processing they require in the following discussions.

2.1.8. Hamming optimality of the next-hop address algorithm. Notice that the specific hypercube routing next-hop computation algorithm explained in **2.1.5** may potentially be replaced by another algorithm, provided it satisfies certain properties. One of these properties is the *Hamming optimality*, meaning that the Hamming (L_1) distance from ξ to η equals the sum of Hamming distances from ξ to $\textsc{NextHop}(\xi,\eta)$ and from $\textsc{NextHop}(\xi,\eta)$ to η:

$$\|\xi-\eta\|_1 = \big\|\xi-\textsc{NextHop}(\xi,\eta)\big\|_1 + \big\|\textsc{NextHop}(\xi,\eta)-\eta\big\|_1 \tag{14}$$

Here $\|\xi-\eta\|_1$ is the *Hamming distance* between ξ and η, equal to the number of bit positions in which ξ and η differ:[15]

$$\|\xi-\eta\|_1 = \sum_i |\xi_i-\eta_i| \tag{15}$$

Notice that in general one should expect only an inequality in (14), following from the triangle inequality for the L_1-metric. Hamming optimality essentially means that $\textsc{NextHop}(\xi,\eta)$ lies on one of the (Hamming) shortest paths from ξ to η. It can also be expressed by saying that $\nu=\textsc{NextHop}(\xi,\eta)$ is always obtained from ξ by changing the values of bits at some positions to their values in η: for any bit position i, we have $\nu_i=\xi_i$ or $\nu_i=\eta_i$.[16]

2.1.9. Non-stopping of NextHop. Another important property of the NextHop is its *non-stopping*, meaning that $\textsc{NextHop}(\xi,\eta)=\xi$ is possible only when $\xi=\eta$. In other words, if we have not yet arrived at η, the next hop cannot coincide with our current position.

This property implies that the path from ξ to η—i.e., the sequence of intermediate addresses $\xi^{(0)}:=\xi$, $\xi^{(n)}:=\textsc{NextHop}(\xi^{(n-1)},\eta)$—will gradually stabilize at η: for some $N\geq 0$, we have $\xi^{(n)}=\eta$ for all $n\geq N$. Indeed, one can always take $N:=\|\xi-\eta\|_1$.

2.1.10. Convexity of the HR path with respect to sharding. A consequence of Hamming optimality property (14) is what we call the *convexity*

[15]When the addresses involved are of different lengths (e.g., because they belong to different workchains), one should consider only the first 96 bits of the addresses in the above formula.

[16]Instead of Hamming optimality, we might have considered the equivalent property of *Kademlia optimality*, written for the Kademlia (or weighted L_1) distance as given by $\|\xi-\eta\|_K:=\sum_i 2^{-i}|\xi_i-\eta_i|$ instead of the Hamming distance.

of the path from ξ to η with respect to sharding. Namely, if $\xi^{(0)} := \xi$, $\xi^{(n)} := \text{NextHop}(\xi^{(n-1)}, \eta)$ is the computed path from ξ to η, and N is the first index such that $\xi^{(N)} = \eta$, and S is a shard of some workchain in any shard configuration, then the indices i with $\xi^{(i)}$ residing in shard S constitute a subinterval in $[0, N]$. In other words, if integers $0 \le i \le j \le k \le N$ are such that $\xi^{(i)}$, $\xi^{(k)} \in S$, then $\xi^{(j)} \in S$ as well.

This convexity property is important for some proofs related to message forwarding in the presence of dynamic sharding.

2.1.11. Internal routing. Notice that the next-hop address computed according to the rules defined in **2.1.5** may belong to the same shardchain as the current one (i.e., the one containing the transit address). In that case, the "internal routing" occurs immediately, the transit address is replaced by the value of the computed next-hop address, and the next-hop address computation step is repeated until a next-hop address lying outside the current shardchain is obtained. The message is then kept in the transit output queue according to its computed next-hop address, with its last computed transit address as the "intermediate owner" of the transit message. If the current shardchain splits into two shardchains before the message is forwarded further, it is the shardchain containing the intermediate owner that inherits this transit message.

Alternatively, we might go on computing the next-hop addresses only to find out that the destination address already belongs to the current shardchain. In that case, the message will be processed (by a transaction) inside this shardchain instead of being forwarded further.

2.1.12. Neighboring shardchains. Two shards in a shard configuration—or the two corresponding shardchains—are said to be *neighbors*, or *neighboring shardchains*, if one of them contains a next-hop address for at least one combination of allowed source and destination addresses, while the other contains the transit address for the same combination. In other words, two shardchains are neighbors if a message can be forwarded directly from one of them into the other via Hypercube Routing.

The masterchain is also included in this definition, as if it were the only shardchain of the workchain with $workchain\_id = -1$. In this respect, it is a neighbor of all the other shardchains.

2.1.13. Any shard is a neighbor of itself. Notice that a shardchain is always considered a neighbor of itself. This may seem redundant, because we

always repeat the next-hop computation described in **2.1.5** until we obtain a next-hop address outside the current shardchain (cf. **2.1.11**). However, there are at least two reasons for such an arrangement:

- Some messages have the source and the destination address inside the same shardchain, at least when the message is created. However, if such a message is not processed immediately in the same block where it has been created, it must be added to the outbound message queue of its shardchain, and be imported as an inbound message (with an entry in the *InMsgDescr*) in one of the subsequent blocks of the same shardchain.[17]

- Alternatively, the next-hop address may originally be in some other shardchain that later gets merged with the current shardchain, so that the next hop becomes inside the same shardchain. Then the message will have to be imported from the outbound message queue of the merged shardchain, and forwarded or processed accordingly to its next-hop address, even though they reside now inside the same shardchain.

2.1.14. Hypercube Routing and the ISP. Ultimately, the Infinite Sharding Paradigm (ISP) applies here: a shardchain should be considered a provisional union of accountchains, grouped together solely to minimize the block generation and transmission overhead.

The forwarding of a message runs through several intermediate accountchains, some of which can happen to lie in the same shard. In this case, once a message reaches an accountchain lying in this shard, it is immediately ("internally") routed inside that shard until the last accountchain lying in the same shard is reached (cf. **2.1.11**). Then the message is enqueued in the output queue of that last accountchain.[18]

2.1.15. Representation of transit and next-hop addresses. Notice that the transit and next-hop addresses differ from the source address only in the *workchain_id* and in the first (most significant) 64 bits of the account address. Therefore, they may be represented by 96-bit strings. Furthermore,

[17] Notice that the next-hop and internal-routing computations are still applied to such messages, since the current shardchain may be split before the message is processed. In this case, the new sub-shardchain containing the destination address will inherit the message.

[18] We may define the (virtual) output queue of an account(chain) as the subset of the *OutMsgQueue* of the shard currently containing that account that consists of messages with transit addresses equal to the address of the account.

their *workchain_id* usually coincides with the *workchain_id* of either the source address or the destination address; a couple of bits may be used to indicate this situation, thus further reducing the space required to represent the transit and next-hop addresses.

In fact, the required storage may be reduced even further by observing that the specific hypercube routing algorithm described in **2.1.5** always generates intermediate (i.e., transit and next-hop) addresses that coincide with the destination address in their first k bits, and with the source address in their remaining bits. Therefore, one might use just the values $0 \leq k_{\rm tr}, k_{\rm nh} \leq 96$ to fully specify the transit and next-hop addresses. One might also notice that $k' := k_{\rm nh}$ turns out to be a fixed function of $k := k_{\rm tr}$ (for instance, $k' = k + n_2 = k + 4$ for $k \geq 32$), and therefore include only one 7-bit value of k in the serialization.

Such optimizations have the obvious disadvantage that they rely too much on the specific routing algorithm used, which can be changed in the future, so they are used in **3.1.15** with a provision to specify more general intermediate addresses if necessary.

2.1.16. Message envelopes. The transit and next-hop addresses of a forwarded message are not included in the message itself, but are kept in a special *message envelope*, which is a cell (or a cell slice) containing the transit and next-hop addresses with the above optimizations, some other information relevant for forwarding and processing, and a reference to a cell containing the unmodified original message. In this way, a message can easily be "extracted" from its original envelope (e.g., the one present in the *InMsgDescr*) and be put into another envelope (e.g., before being included into the *OutMsgQueue*).

In the representation of a block as a tree, or rather a DAG, of cells, the two different envelopes will contain references to a shared cell with the original message. If the message is large, this arrangement avoids the need to keep more than one copy of the message in the block.

2.2 Hypercube Routing protocol

This section exposes the details of the hypercube routing protocol employed by the TON Blockchain to achieve guaranteed delivery of messages between smart contracts residing in arbitrary shardchains. For the purposes of this document, we will refer to the variant of hypercube routing employed by the

TON Blockchain as Hypercube Routing (HR).

2.2.1. Message uniqueness. Before continuing, let us observe that any (internal) message is *unique*. Recall that a message contains its full source address along with its logical creation time, and all outbound messages created by the same smart contract have strictly increasing logical creation times (cf. **1.4.6**); therefore, the combination of the full source address and the logical creation time uniquely defines the message. Since we assume the chosen hash function SHA256 to be collision resistant, *a message is uniquely determined by its hash*, so we can identify two messages if we know that their hashes coincide.

This does not extend to external messages "from nowhere", which have no source addresses. Special care must be taken to prevent replay attacks related to such messages, especially by designers of user wallet smart contracts. One possible solution is to include a sequence number in the body of such messages, and keep the count of external messages already processed inside the smart-contract persistent data, refusing to process an external message if its sequence number differs from this count.

2.2.2. Identifying messages with equal hashes. The TON Blockchain assumes that two messages with the same hashes coincide, and treats either of them as a redundant copy of the other. As explained above in **2.2.1**, this does not lead to any unexpected effects for internal messages. However, if one sends two coinciding "messages from nowhere" to a smart contract, it may happen that only one of them will be delivered—or both. If their action is not supposed to be idempotent (i.e., if processing the message twice has a different effect from processing it once), some provisions should be made to distinguish the two messages, for instance by including a sequence number in them.

In particular, the *InMsgDescr* and *OutMsgDescr* use the (unenveloped) message hash as a key, tacitly assuming that distinct messages have distinct hashes. In this way, one can trace the path and the fate of a message across different shardchains by looking up the message hash in the *InMsgDescr* and *OutMsgDescr* of different blocks.

2.2.3. The structure of *OutMsgQueue*. Recall that the outbound messages — both those created inside the shardchain, and transit messages previously imported from a neighboring shardchain to be relayed to the next-hop shardchain — are accumulated in the *OutMsgQueue*, which is part of the

state of the shardchain (cf. **1.2.7**). In contrast with *InMsgDescr* and *OutMsgDescr*, the key in *OutMsgQueue* is not the message hash, but its next-hop address—or at least its first 96 bits—concatenated with the message hash.

Furthermore, the *OutMsgQueue* is not just a dictionary (hashmap), mapping its keys into (enveloped) messages. Rather, it is a *min-augmented dictionary with respect to the logical creation time*, meaning that each node of the Patricia tree representing *OutMsgQueue* has an additional value (in this case, an unsigned 64-bit integer), and that this augmentation value in each fork node is set to be equal to the minimum of the augmentation values of its children. The augmentation value of a leaf equals the logical creation time of the message contained in that leaf; it need not be stored explicitly.

2.2.4. Inspecting the *OutMsgQueue* of a neighbor. Such a structure for the *OutMsgQueue* enables the validators of a neighboring shardchain to inspect it to find its part (Patricia subtree) relevant to them (i.e., consisting of messages with the next-hop address belonging to the neighboring shard in question—or having the next-hop address with a given binary prefix), as well as quickly compute the "oldest" (i.e., with the minimum logical creation time) message in that part.

Furthermore, the shard validators do not even need to track the total state of all their neighboring shardchains—they only need to keep and update a copy of their *OutMsgQueue*, or even of its subtree related to them.

2.2.5. Logical time monotonicity: importing the oldest message from the neighbors. The first fundamental local condition of message forwarding, called *(message import) (logical time) monotonicity condition*, may be summarized as follows:

> While importing messages into the *InMsgDescr* of a shardchain block from the *OutMsgQueue*s of its neighboring shardchains, the validators must import the messages in the increasing order of their logical time; in the case of a tie, the message with the smaller hash is imported first.

More precisely, each shardchain block contains the hash of a masterchain block (assumed to be "the latest" masterchain block at the time of the shardchain block's creation), which in turn contains the hashes of the most recent shardchain blocks. In this way, each shardchain block indirectly "knows"

the most recent state of all other shardchains, and especially its neighboring shardchains, including their *OutMsgQueue*s.[19]

Now an alternative equivalent formulation of the monotonicity condition is as follows:

> If a message is imported into the *InMsgDescr* of the new block, its logical creation time cannot be greater than that of any message left unimported in the *OutMsgQueue* of the most recent state of any of the neighboring shardchains.

It is this form of the monotonicity condition that appears in the local consistency conditions of the TON Blockchain blocks and is enforced by the validators.

2.2.6. Witnesses to violations of the message import logical time monotonicity condition. Notice that if this condition is not fulfilled, a small Merkle proof witnessing its failure may be constructed. Such a proof will contain:

- A path in the *OutMsgQueue* of a neighbor from the root to a certain message m with small logical creation time.
- A path in the *InMsgDescr* of the block under consideration showing that the key equal to $\text{Hash}(m)$ is absent in *InMsgDescr* (i.e., that m has not been included in the current block).
- A proof that m has not been included in a preceding block of the same shardchain, using the block header information containing the smallest and the largest logical time of all messages imported into the block (cf. **2.3.4**–**2.3.7** for more information).
- A path in *InMsgDescr* to another included message m', such that either $\text{Lt}(m') > \text{Lt}(m)$, or $\text{Lt}(m') = \text{Lt}(m)$ and $\text{Hash}(m') > \text{Hash}(m)$.

2.2.7. Deleting a message from *OutMsgQueue*. A message must be deleted from *OutMsgQueue* sooner or later; otherwise, the storage used by *OutMsgQueue* would grow to infinity. To this end, several "garbage collection

[19] In particular, if the hash of a recent block of a neighboring shardchain is not yet reflected in the latest masterchain block, its modifications to *OutMsgQueue* must not be taken into account.

rules" are introduced. They allow the deletion of a message from *OutMsgQueue* during the evaluation of a block only if an explicit special "delivery record" is present in the *OutMsgDescr* of that block. This record contains either a reference to the neighboring shardchain block that has included the message into its *InMsgDescr* (the hash of the block is sufficient, but collated material for the block may contain the relevant Merkle proof), or a Merkle proof of the fact that the message has been delivered to its final destination via Instant Hypercube Routing.

2.2.8. Guaranteed message delivery via Hypercube Routing. In this way, a message cannot be deleted from the outbound message queue unless it has been either relayed to its next-hop shardchain or delivered to its final destination (cf. **2.2.7**). Meanwhile, the message import monotonicity condition (cf. **2.2.5**) ensures that any message will sooner or later be relayed into the next shardchain, taking into account other conditions which require the validators to use at least half of the block's space or gas limits for importing inbound internal messages (otherwise the validators might choose to create empty blocks or import only external messages even in the presence of non-empty outbound message queues at their neighbors).

2.2.9. Message processing order. When several imported messages are processed by transactions inside a block, the *message processing order conditions* ensure that older messages are processed first. More precisely, if a block contains two transactions t and t' of the same account, which process inbound messages m and m', respectively, and $\text{LT}(m) < \text{LT}(m')$, then we must have $\text{LT}(t) < \text{LT}(t')$.

2.2.10. FIFO guarantees of Hypercube Routing. The message processing order conditions (cf. **2.2.9**), along with the message import monotonicity conditions (cf. **2.2.5**), imply the *FIFO guarantees for Hypercube Routing*. Namely, if a smart contract ξ creates two messages m and m' with the same destination η, and m' is generated later than m (meaning that $m \prec m'$, hence $\text{LT}(m) < \text{LT}(m')$), then m will be processed by η before m'. This is so because both messages will follow the same routing steps on the path from ξ to η (the Hypercube Routing algorithm described in **2.1.5** is deterministic), and in all outbound queues and inbound message descriptions m' will appear "after" m.[20]

[20]This statement is not as trivial as it seems at first, because some of the shardchains

If message m' can be delivered to B via Instant Hypercube Routing, this is not necessarily true anymore. Therefore, a simple way of ensuring FIFO message delivery discipline between a pair of smart contracts consists in setting a special bit in the message header preventing its delivery via IHR.

2.2.11. Delivery uniqueness guarantees of Hypercube Routing. Notice that the message import monotonicity conditions also imply the *uniqueness* of the delivery of any message via Hypercube Routing—i.e., that it cannot be imported and processed by the destination smart contract more than once. We will see later in **2.3** that enforcing delivery uniqueness when both Hypercube Routing and Instant Hypercube Routing are active is more complicated.

2.2.12. An overview of Hypercube Routing. Let us summarize all routing steps performed to deliver an internal message m created by source account ξ_0 to destination account η. We denote by $\xi_{k+1} := \textsc{NextHop}(\xi_k, \eta)$, $k = 0, 1, 2, \ldots$ the intermediate addresses dictated by HR for forwarding the message m to its final destination η. Let S_k be the shard containing ξ_k.

- [Birth] — Message m with destination η is created by a transaction t belonging to an account ξ_0 residing in some shardchain S_0. The logical creation time $\textsc{Lt}(m)$ is fixed at this point and included into the message m.

- [ImmediateProcessing?] — If the destination η resides in the same shardchain S_0, the message may be processed in the same block it was generated in. In this case, m is included into *OutMsgDescr* with a flag indicating it has been processed in this very block and need not be forwarded further. Another copy of m is included into *InMsgDescr*, along with the usual data describing the processing of inbound messages. (Notice that m is not included into the *OutMsgQueue* of S_0.)

involved may split or merge during the routing. A correct proof may be obtained by adopting the ISP perspective to HR as explained in **2.1.14** and observing that m' will always be behind m, either in terms of the intermediate accountchain reached or, if they happen to be in the same accountchain, in terms of logical creation time.

A crucial observation is that "at any given moment of time" (logically; a more precise description would be "in the total state obtained after processing any causally closed subset $\mathscr{F}$ of blocks"), the intermediate accountchains belonging to the same shard are contiguous on the path from ξ to η (i.e., cannot have accountchains belonging to some other shard in between). This is a "convexity property" (cf. **2.1.10**) of the Hypercube Routing algorithm described in **2.1.5**.

- [InitialInternalRouting] — If m either has a destination outside S_0, or is not processed in the same block where it was generated, the internal routing procedure described in **2.1.11** is applied, until an index k is found such that ξ_k lies in S_0, but $\xi_{k+1} = \text{NextHop}(\xi_k, \eta)$ does not (i.e., $S_k = S_0$, but $S_{k+1} \neq S_0$). Alternatively, this process stops if $\xi_k = \eta$ or ξ_k coincides with η in its first 96 bits.

- [OutboundQueuing] — The message m is included into *OutMsgDescr* (with the key equal to its hash), with an envelope containing its transit address ξ_k and next-hop address ξ_{k+1} as explained in **2.1.16** and **2.1.15**. The same enveloped message is also included in the *OutMsgQueue* of the state of S_k, with the key equal to the concatenation of the first 96 bits of its next-hop address ξ_{k+1} (which may be equal to η if η belongs to S_k) and the message hash $\text{Hash}(m)$.

- [QueueWait] — Message m waits in the *OutMsgQueue* of shardchain S_k to be forwarded further. In the meantime, shardchain S_k may split or merge with other shardchains; in that case, the new shard S'_k containing the transit address ξ_k inherits m in its *OutMsgQueue*.

- [ImportInbound] — At some point in the future, the validators for the shardchain S_{k+1} containing the next-hop address ξ_{k+1} scan the *OutMsgQueue* in the state of shardchain S_k and decide to import message m in keeping with the monotonicity condition (cf. **2.2.5**) and other conditions. A new block for shardchain S_{k+1} is generated, with an enveloped copy of m included in its *InMsgDescr*. The entry in *InMsgDescr* contains also the *reason* for importing m into this block, with a hash of the most recent block of shardchain S'_k, and the previous next-hop and transit addresses ξ_k and ξ_{k+1}, so that the corresponding entry in the *OutMsgQueue* of S'_k can be easily located.

- [Confirmation] — This entry in the *InMsgDescr* of S_{k+1} also serves as a confirmation for S'_k. In a later block of S'_k, message m must be removed from the *OutMsgQueue* of S'_k; this modification is reflected in a special entry in the *OutMsgDescr* of the block of S'_k that performs this state modification.

- [Forwarding?] — If the final destination η of m does not reside in S_{k+1}, the message is *forwarded*. Hypercube Routing is applied until

some ξ_l, $l > k$, and $\xi_{l+1} = \text{NextHop}(\xi_l, \eta)$ are obtained, such that ξ_l lies in S_{k+1}, but ξ_{l+1} does not (cf. **2.1.11**). After that, a newly-enveloped copy of m with transit address set to ξ_l and next-hop address ξ_{l+1} is included into both the *OutMsgDescr* of the current block of S_{k+1} and the *OutMsgQueue* of the new state of S_{k+1}. The entry of m in *InMsgDescr* contains a flag indicating that the message has been forwarded; the entry in *OutMsgDescr* contains the newly-enveloped message and a flag indicating that this is a forwarded message. Then all the steps starting from [OutboundQueueing] are repeated, for l instead of k.

- [Processing?] — If the final destination η of m resides in S_{k+1}, then the block of S_{k+1} that imported the message must process it by a transaction t included in the same block. In this case, *InMsgDescr* contains a reference to t by its logical time $\text{Lt}(t)$, and a flag indicating that the message has been processed.

The above message routing algorithm does not take into account some further modifications required to implement Instant Hypercube Routing (IHR). For instance, a message may be *discarded* after being imported (listed in *InMsgDescr*) into its final or intermediate shardchain block, because a proof of delivery via IHR to the final destination is presented. In this case, such a proof must be included into *InMsgDescr* to explain why the message was not forwarded further or processed.

2.3 Instant Hypercube Routing and combined delivery guarantees

This section describes the Instant Hypercube Routing protocol, normally applied by TON Blockchain in parallel to the previously discussed Hypercube Routing protocol to achieve faster message delivery. However, when both Hypercube Routing and Instant Hypercube Routing are applied to the same message in parallel, achieving delivery and unique delivery guarantees is more complicated. This topic is also discussed in this section.

2.3.1. An overview of Instant Hypercube Routing. Let us explain the major steps applied when the Instant Hypercube Routing (IHR) mechanism is applied to a message. (Notice that normally both the usual HR and IHR

work in parallel for the same message; some provisions must be taken to guarantee the uniqueness of delivery of any message.)

Consider the routing and delivery of the same message m with source ξ and destination η as discussed in **2.2.12**:

- [NetworkSend] — After the validators of S_0 have agreed on and signed the block containing the creating transaction t for m, and observed that the destination η of m does not reside inside S_0, they may send a datagram (encrypted network message), containing the message m along with a Merkle proof of its inclusion into the *OutMsgDescr* of the block just generated, to the validator group of the shardchain T currently owning the destination η.

- [NetworkReceive] — If the validators of shardchain T receive such a message, they check its validity starting from the most recent masterchain block and the shardchain block hashes listed in it, including the most recent "canonical" block of shardchain S_0 as well. If the message is invalid, they silently discard it. If that block of shardchain S_0 has a larger sequence number than the one listed in the most recent masterchain block, they may either discard it or postpone the verification until the next masterchain block appears.

- [InclusionConditions] — The validators check inclusion conditions for message m. In particular, they must check that this message has not been delivered before, and that the *OutMsgQueue*s of the neighbors do not have unprocessed outbound messages with destinations in T with smaller logical creation times than $\text{Lt}(m)$.

- [Deliver] — The validators deliver and process the message, by including it into the *InMsgDescr* of the current shardchain block along with a bit indicating that it is an IHR message, the Merkle proof of its inclusion into the *OutMsgDescr* of the original block, and the logical time of the transaction t' processing this inbound message into the currently generated block.

- [Confirm] — Finally, the validators send encrypted datagrams to all the validator groups of the intermediate shardchains on the path from ξ to η, containing a Merkle proof of the inclusion of message m into the *InMsgDescr* of its final destination. The validators of an intermediate shardchain may use this proof to *discard* the copy of message

m travelling by the rules of HR, by importing the message into their *InMsgDescr* along with the Merkle proof of final delivery and setting a flag indicating that the message has been discarded.

The overall procedure is even simpler than that for Hypercube Routing. Notice, however, that IHR comes with no delivery or FIFO guarantees: the network datagram may be lost in transit, or the validators of the destination shardchain may decide not to act on it, or they may discard it due to buffer overflow. This is the reason why IHR is used as a complement to HR, and not as a replacement.

2.3.2. Overall eventual delivery guarantees. Notice that the combination of HR and IHR guarantees the ultimate delivery of any internal message to its final destination. Indeed, the HR by itself is guaranteed to deliver any message eventually, and the HR for message m can be cancelled at an intermediate stage only by a Merkle proof of delivery of m to its final destination (via IHR).

2.3.3. Overall unique delivery guarantees. However, the *uniqueness* of message delivery for the combination of HR and IHR is more difficult to achieve. In particular, one must check the following conditions, and, if necessary, be able to provide short Merkle proofs that they do or don't hold:

- When a message m is imported into its next intermediate shardchain block via HR, we must check that m has not already been imported via HR.
- When m is imported and processed in its final destination shardchain, we must check that m has not already been processed. If it has, there are three subcases:
  - If m is being considered for import via HR, and it has already been imported via HR, it must not be imported at all.
  - If m is being considered for import via HR, and it has already been imported via IHR (but not HR), then it must be imported and immediately discarded (without being processed by a transaction). This is necessary to remove m from the *OutMsgQueue* of its previous intermediate shardchain.

– If m is being considered for import via IHR, and it has already been imported via either IHR or HR, it must not be imported at all.

2.3.4. Checking whether a message has already been delivered to its final destination. Consider the following general algorithm for checking whether a message m has already been delivered to its final destination η: One can simply scan the last several blocks belonging to the shardchain containing the destination address, starting from the latest block and working backwards through the previous block references. (If there are two previous blocks—i.e., if a shardchain merge event occurred at some point—one would follow the chain containing the destination address.) The *InMsgDescr* of each of these blocks can be checked for an entry with key $\textsc{Hash}(m)$. If such an entry is found, the message m has already been delivered, and we can easily construct a Merkle proof of this fact. If we do not find such an entry before arriving at a block B with $\textsc{Lt}^+(B) < \textsc{Lt}(m)$, implying that m could not be delivered in B or any of its predecessors, then the message m definitely has not been delivered yet.

The obvious disadvantage of this algorithm is that, if message m is very old (and most likely delivered a long time ago), meaning that it has a small value of $\textsc{Lt}(m)$, then a large number of blocks will need to be scanned before yielding an answer. Furthermore, if the answer is negative, the size of the Merkle proof of this fact will increase linearly with the number of blocks scanned.

2.3.5. Checking whether an IHR message has already been delivered to its final destination. To check whether an IHR message m has already been delivered to its destination shardchain, we can apply the general algorithm described above (cf. **2.3.4**), modified to inspect only the last c blocks for some small constant c (say, $c = 8$). If no conclusion can be reached after inspecting these blocks, then the validators for the destination shardchain may simply discard the IHR message instead of spending more resources on this check.

2.3.6. Checking whether an HR message has already been delivered via HR to its final destination or an intermediate shardchain. To check whether an HR-received message m (or rather, a message m being considered for import via HR) has already been imported via HR, we can use the following algorithm: Let ξ_k be the transit address of m (belonging to

a neighboring shardchain S_k) and ξ_{k+1} be its next-hop address (belonging to the shardchain under consideration). Since we are considering the inclusion of m, m must be present in the *OutMsgQueue* of the most recent state of shardchain S_k, with ξ_k and ξ_{k+1} indicated in its envelope. In particular, (a) the message has been included into *OutMsgQueue*, and we may even know when, because the entry in *OutMsgQueue* sometimes contains the logical time of the block where it has been added, and (b) it has not yet been removed from *OutMsgQueue*.

Now, the validators of the neighboring shardchain are required to remove a message from *OutMsgQueue* as soon as they observe that message (with transit and next-hop addresses ξ_k and ξ_{k+1} in its envelope) has been imported into the *InMsgDescr* of the message's next-hop shardchain. Therefore, (b) implies that the message could have been imported into the *InMsgDescr* of a preceding block only if this preceding block is very new (i.e., not yet known to the most recent neighboring shardchain block). Therefore, only a very limited number of preceding blocks (typically one or two, at most) need to be scanned by the algorithm described in **2.3.4** to conclude that the message has not yet been imported.[21] In fact, if this check is performed by the validators or collators for the current shardchain themselves, it can be optimized by keeping in memory the *InMsgDescr*s of the several latest blocks.

2.3.7. Checking whether an HR message has already been delivered via IHR to its final destination. Finally, to check whether an HR message has already been delivered to its final destination via IHR, one can use the general algorithm described in **2.3.4**. In contrast with **2.3.5**, we cannot abort the verification process after scanning a fixed number of the latest blocks in the destination shardchain, because HR messages cannot be dropped without a reason.

Instead, we indirectly bound the number of blocks to be inspected by forbidding the inclusion of IHR message m into a block B of its destination shardchain if there are already more than, say, $c = 8$ blocks B' in the destination shardchain with $\text{LT}^+(B') \geq \text{LT}(m)$.

Such a condition effectively restricts the time interval after the creation of message m in which it could have been delivered via IHR, so that only a small number of blocks of the destination shardchain (at most c) will need

[21] One must not only look up the key $\text{HASH}(m)$ in the *InMsgDescr* of these blocks, but also check the intermediate addresses in the envelope of the corresponding entry, if found.

to be inspected.

Notice that this condition nicely aligns with the modified algorithm described in **2.3.5**, effectively forbidding the validators from importing the newly-received IHR message if more than $c = 8$ steps are needed to check that it had not been imported already.

3 Messages, message descriptors, and queues

This chapter presents the internal layout of individual messages, message descriptors (such as *InMsgDescr* or *OutMsgDescr*), and message queues (such as *OutMsgQueue*). Enveloped messages (cf. **2.1.16**) are also discussed here.

Notice that most general conventions related to messages must be obeyed by all shardchains, even if they do not belong to the basic shardchain; otherwise, messaging and interaction between different workchains would not be possible. It is the *interpretation* of the message contents and the *processing* of messages, usually by some transactions, that differs between workchains.

3.1 Address, currency, and message layout

This chapter begins with some general definitions, followed by the precise layout of addresses used for serializing source and destination addresses in a message.

3.1.1. Some standard definitions. For the reader's convenience, we reproduce here several general TL-B definitions.[22] These definitions are used below in the discussion of address and message layout, but otherwise are not related to the TON Blockchain.

```
unit$_ = Unit;
true$_ = True;
// EMPTY False;
bool_false$0 = Bool;
bool_true$1 = Bool;
nothing$0 {X:Type} = Maybe X;
just$1 {X:Type} value:X = Maybe X;
left$0 {X:Type} {Y:Type} value:X = Either X Y;
right$1 {X:Type} {Y:Type} value:Y = Either X Y;
pair$_ {X:Type} {Y:Type} first:X second:Y = Both X Y;

bit$_ _:(## 1) = Bit;
```

3.1.2. TL-B scheme for addresses. The serialization of source and destination addresses is defined by the following TL-B scheme:

[22] A description of an older version of TL may be found at `https://core.telegram.org/mtproto/TL`. Alternatively, an informal introduction to TL-B schemes may be found in [4, 3.3.4].


```
addr_none$00 = MsgAddressExt;
addr_extern$01 len:(## 8) external_address:(len * Bit)
             = MsgAddressExt;
anycast_info depth:(## 5) rewrite_pfx:(depth * Bit) = Anycast;
addr_std$10 anycast:(Maybe Anycast)
   workchain_id:int8 address:uint256  = MsgAddressInt;
addr_var$11 anycast:(Maybe Anycast) addr_len:(## 9)
   workchain_id:int32 address:(addr_len * Bit) = MsgAddressInt;
_ MsgAddressInt = MsgAddress;
_ MsgAddressExt = MsgAddress;
```

The two last lines define type `MsgAddress` to be the internal union of types `MsgAddressInt` and `MsgAddressExt` (not to be confused with their external union `Either MsgAddressInt MsgAddressExt` as defined in **3.1.1**), as if the preceding four lines had been repeated with the right-hand side replaced by `MsgAddress`. In this way, type `MsgAddress` has four constructors, and types `MsgAddressInt` and `MsgAddressExt` are both subtypes of `MsgAddress`.

3.1.3. External addresses. The first two constructors, `addr_none` and `addr_extern`, are used for source addresses of "messages from nowhere" (inbound external messages), and for destination addresses of "messages to nowhere" (outbound external messages). The `addr_extern` constructor defines an "external address", which is ignored by the TON Blockchain software altogether (which treats `addr_extern` as a longer variant of `addr_none`), but may be used by external software for its own purposes. For example, a special external service may inspect the destination address of all outbound external messages found in all blocks of the TON Blockchain, and, if a special magic number is present in the `external_address` field, parse the remainder as an IP address and UDP port or a (TON Network) ADNL address, and send a datagram with a copy of the message to the network address thus obtained.

3.1.4. Internal addresses. The two remaining constructors, `addr_std` and `addr_var`, represent internal addresses. The first of them, `addr_std`, represents a signed 8-bit *workchain_id* (sufficient for the masterchain and for the basic workchain) and a 256-bit internal address in the selected workchain. The second of them, `addr_var`, represents addresses in workchains with a "large" *workchain_id*, or internal addresses of length not equal to 256. Both of these constructors have an optional `anycast` value, absent by default,

which enables "address rewriting" when present.[23]

The validators must use `addr_std` instead of `addr_var` whenever possible, but must be ready to accept `addr_var` in inbound messages. The `addr_var` constructor is intended for future extensions.

Notice that *workchain_id* must be a valid workchain identifier enabled in the current masterchain configuration, and the length of the internal address must be in the range allowed for the indicated workchain. For example, one cannot use *workchain_id* $= 0$ (basic workchain) or *workchain_id* $= -1$ (masterchain) with addresses that are not exactly 256 bits long.

3.1.5. Representing Gram currency amounts. Amounts of Grams are expressed with the aid of two types representing variable-length unsigned or signed integers, plus a type `Grams` explicitly dedicated to representing non-negative amounts of nanograms, as follows:

```
var_uint$_ {n:#} len:(#< n) value:(uint (len * 8))
         = VarUInteger n;
var_int$_ {n:#} len:(#< n) value:(int (len * 8))
        = VarInteger n;
nanograms$_ amount:(VarUInteger 16) = Grams;
```

If one wants to represent x nanograms, one selects an integer $\ell < 16$ such that $x < 2^{8\ell}$, and serializes first ℓ as an unsigned 4-bit integer, then x itself as an unsigned 8ℓ-bit integer. Notice that four zero bits represent a zero amount of Grams.

Recall (cf. [3, A]) that the original total supply of Grams is fixed at five billion (i.e., $5 \cdot 10^{18} < 2^{63}$ nanograms), and is expected to grow very slowly. Therefore, all the amounts of Grams encountered in practice will fit in unsigned or even signed 64-bit integers. The validators may use the 64-bit integer representation of Grams in their internal computations; however, the serialization of these values the blockchain is another matter.

3.1.6. Representing collections of arbitrary currencies. Recall that the TON Blockchain allows its users to define arbitrary cryptocurrencies

[23] *Address rewriting* is a feature used to implement "anycast addresses" employed by the so-called *large* or *global* smart contracts (cf. [3, 2.3.18]), which can have instances in several shardchains. When address rewriting is enabled, a message may be routed to and processed by a smart contract with an address coinciding with the destination address up to the first d bits, where $d \leq 32$ is the "splitting depth" of the smart contract indicated in the `anycast.depth` field (cf. **2.1.7**). Otherwise, the addresses must match exactly.

or tokens apart from the Gram, provided some conditions are met. Such additional cryptocurrencies are identified by 32-bit *currency_ id*s. The list of defined additional cryptocurrencies is a part of the blockchain configuration, stored in the masterchain.

When some amounts of one or several such cryptocurrencies need to be represented, a dictionary (cf. [4, 3.3]) with 32-bit *currency_ id*s as keys and `VarUInteger 32` values is used:

```
extra_currencies$_ dict:(HashMapE 32 (VarUInteger 32))
                 = ExtraCurrencyCollection;
currencies$_ grams:Grams other:ExtraCurrencyCollection
           = CurrencyCollection;
```

The value attached to an internal message is represented by a value of the `CurrencyCollection` type, which may describe a certain (non-negative) amount of (nano)grams as well as some additional currencies, if needed. Notice that if no additional currencies are required, `other` reduces to just one zero bit.

3.1.7. Message layout. A message consists of its *header* followed by its *body*, or *payload*. The body is essentially arbitrary, to be interpreted by the destination smart contract. The message header is standard and is organized as follows:

```
int_msg_info$0 ihr_disabled:Bool bounce:Bool
  src:MsgAddressInt dest:MsgAddressInt
  value:CurrencyCollection ihr_fee:Grams fwd_fee:Grams
  created_lt:uint64 created_at:uint32 = CommonMsgInfo;
ext_in_msg_info$10 src:MsgAddressExt dest:MsgAddressInt
  import_fee:Grams = CommonMsgInfo;
ext_out_msg_info$11 src:MsgAddressInt dest:MsgAddressExt
  created_lt:uint64 created_at:uint32 = CommonMsgInfo;

tick_tock$_ tick:Boolean tock:Boolean = TickTock;

_ split_depth:(Maybe (## 5)) special:(Maybe TickTock)
  code:(Maybe ^Cell) data:(Maybe ^Cell)
  library:(Maybe ^Cell) = StateInit;
```

```
message$_ {X:Type} info:CommonMsgInfo
  init:(Maybe (Either StateInit ^StateInit))
  body:(Either X ^X) = Message X;
```

The meaning of this scheme is as follows.

Type `Message` X describes a message with the body (or payload) of type X. Its serialization starts with `info` of type `CommonMsgInfo`, which comes in three flavors: for internal messages, inbound external messages, and outbound external messages, respectively. All of them have a source address `src` and destination address `dest`, which are external or internal according to the chosen constructor. Apart from that, an internal message may bear some `value` in Grams and other defined currencies, and all messages generated inside the TON Blockchain have a logical creation time `created_lt` (cf. **1.4.6**) and creation unixtime `created_at`, both automatically set by the generating transaction. The creation unixtime equals the creation unixtime of the block containing the generating transaction.

3.1.8. Forwarding and IHR fees. Total value of an internal message. Internal messages define an `ihr_fee` in Grams, which is subtracted from the value attached to the message and awarded to the validators of the destination shardchain if they include the message by the IHR mechanism. The `fwd_fee` is the original total forwarding fee paid for using the HR mechanism; it is automatically computed from some configuration parameters and the size of the message at the time the message is generated.

Notice that the total value carried by a newly-created internal outbound message equals the sum of `value`, `ihr_fee`, and `fwd_fee`. This sum is deducted from the balance of the source account. Of these components, only `value` is always credited to the destination account on message delivery. The `fwd_fee` is collected by the validators on the HR path from the source to the destination, and the `ihr_fee` is either collected by the validators of the destination shardchain (if the message is delivered via IHR), or credited to the destination account.

3.1.9. Code and data portions contained in a message. Apart from the common message information stored in `info`, a message can contain portions of the destination smart contract's code and data. This feature is used, for instance, in the so-called *constructor messages* (cf. **1.7.3**), which are simply internal or inbound external messages with `code` and possibly `data` fields defined in their `init` portions. If the hash of these fields is correct, and the

destination smart contract has no code or data, the values from the message are used instead.[24]

3.1.10. Using `code` and `data` for other purposes. Workchains other than the masterchain and the basic workchain are free to use the trees of cells referred to in the `code`, `data`, and `library` fields for their own purposes. The messaging system itself makes no assumptions about their contents; they become relevant only when a message is processed by a transaction.

3.1.11. Absence of an explicit gas price and gas limit. Notice that messages do not have an explicit gas price and gas limit. Instead, the gas price is set globally by the validators for each workchain (it is a special configurable parameter), and the gas limit for each transaction has also a default value, which is a configurable parameter; the smart contract itself may lower the gas limit during its execution if so desired.

For internal messages, the initial gas limit cannot exceed the Gram value of the message divided by the current gas price. For inbound external messages, the initial gas limit is very small, and the true gas limit is set by the receiving smart contract itself, when it *accepts* the inbound message by the corresponding TVM primitive.

3.1.12. Deserialization of a message payload. The payload, or body, of a message is deserialized by the receiving smart contract when executed by TVM. The messaging system itself makes no assumptions about the internal format of the payload. However, it makes sense to describe the serialization of supported inbound messages by TL or TL-B schemes with 32-bit constructor tags, so that the developers of other smart contracts will know the interface supported by a specific smart contract.

A message is always serialized inside the blockchain as the last field in a cell. Therefore, the blockchain software may assume that whatever bits and references left unparsed after parsing the fields of a `Message` preceding `body` belong to the payload `body` : X, without knowing anything about the serialization of the type X.

[24] More precisely, the information from the `init` field of an inbound message is used either when the receiving account is uninitialized or frozen with the hash of *StateInit* equal to the one expected by the account, or when the receiving account is active, and its code or data is an external hash reference matching the hash of the code or data received in the *StateInit* of the message.

3.1.13. Messages with empty payloads. The payload of a message may happen to be an empty cell slice, having no data bits and no references. By convention, such messages are used for simple value transfers. The receiving smart contract is normally expected to process such messages quietly and to terminate successfully (with a zero exit code), although some smart contracts may perform non-trivial actions even when receiving a message with empty payload. For example, a smart contract may check the resulting balance, and, if it becomes sufficient for a previously postponed action, trigger this action. Alternatively, the smart contract might want to remember in its persistent storage the amount received and the corresponding sender, in order, for instance, to distribute some tokens later to each sender proportionally to the funds transferred.

Notice that even if a smart contract makes no special provisions for messages with empty payloads and throws an exception while processing such messages, the received value (minus the gas payment) will still be added to the balance of the smart contract.

3.1.14. Message source address and logical creation time determine its generating block. Notice that *the source address and the logical creation time of an internal or an outbound external message uniquely determine the block in which the message has been generated*. Indeed, the source address determines the source shardchain, and the blocks of this shardchain are assigned non-intersecting logical time intervals, so only one of them may contain the indicated logical creation time. This is the reason why no explicit mention of the generating block is needed in messages.

3.1.15. Enveloped messages. *Message envelopes* are used for attaching routing information, such as the current (transit) address and the next-hop address, to inbound, transit, and outbound messages (cf. **2.1.16**). The message itself is kept in a separate cell and referred to from the message envelope by a cell reference.

```
interm_addr_regular$0 use_src_bits:(#<= 96)
  = IntermediateAddress;
interm_addr_simple$10 workchain_id:int8 addr_pfx:(64 * Bit)
  = IntermediateAddress;
interm_addr_ext$11 workchain_id:int32 addr_pfx:(64 * Bit)
  = IntermediateAddress;
msg_envelope cur_addr:IntermediateAddress
```

```
next_addr:IntermediateAddress fwd_fee_remaining:Grams
msg:^Message = MsgEnvelope;
```

The `IntermediateAddress` type is used to describe the intermediate addresses of a message—that is, its current (or transit) address `cur_addr`, and its next-hop address `next_addr`. The first constructor `interm_addr_regular` represents the intermediate address using the optimization described in **2.1.15**, by storing the number of the first bits of the intermediate address that are the same as in the source address; the two other explicitly store the workchain identifier and the first 64 bits of the address inside that workchain (the remaining bits can be taken from the source address). The `fwd_fee_remaining` field is used to explicitly represent the maximum amount of message forwarding fees that can be deducted from the message value during the remaining HR steps; it cannot exceed the value of `fwd_fee` indicated in the message itself.

3.2 Inbound message descriptors

This section discusses *InMsgDescr*, the structure containing a description of all inbound messages imported into a block.[25]

3.2.1. Types and sources of inbound messages. Each inbound message mentioned in *InMsgDescr* is described by a value of type *InMsg* (an "inbound message descriptor"), which specifies the source of the message, the reason for its being imported into this block, and some information about its "fate"—its processing by a transaction or forwarding inside the block.

Inbound messages may be classified as follows:

- *Inbound external messages* — Need no additional reason for being imported into the block, but must be immediately processed by a transaction in the same block.
- *Internal IHR messages with destination addresses in this block* — The reason for their being imported into the block includes a Merkle proof of their generation (i.e., their inclusion in *OutMsgDescr* of their original block). Such a message must be immediately delivered to its final destination and processed by a transaction.

[25]Strictly speaking, *InMsgDescr* is the *type* of this structure; we deliberately use the same notation to describe the only instance of this type in a block.

